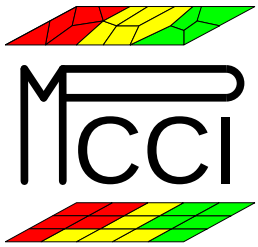

MpCCI

Mesh-based parallel Code Coupling Interface

User's Guide of MpCCI Version 2.0

Fraunhofer Institut
Algorithmen und Wissen-
schaftliches Rechnen

Mesh-based parallel Code Coupling Interface

User's Guide of MpCCI *Version 2.0*

Fraunhofer Institute for Algorithms and Scientific Computing SCAI
Schloss Birlinghoven, 53754 Sankt Augustin, Germany

October 23, 2003

Preface

The growing need for merging different numerical codes for coupled computations in industry and science established the idea to develop a generic coupling tool independent of any special application.

The design of the MpCCI code coupling interface is based on the effort of two predecessors, the coupling interface GRISSLi-CI, funded by the BMB+F project GRISSLi, and the coupling communication library COCOLIB, developed during the EU funded project CISPAR. The work of all people involved in these projects went into the design and implementation of MpCCI and is gratefully acknowledged. The current development is partly funded by the German Federal Ministery of Education and Research (BMB+F) in the project COSIWIT, contract number 01AK950-A .

The specification of MpCCI as well as the programming work was carried out at the Fraunhofer Institute for Algorithms and Scientific Computing (SCAI).

We hope that this software package is of some relevance for your work and will help you to find a new approach to the numerical, technical or scientific issues in question. For further information, the downloadable library itself and recent announcements the reader is asked to visit the MpCCI web site http://www.mpcci.org/.

The MpCCI team:

Regine Ahrem,
Hanno Baehr,
Axel Bürgers,
Andrea Holtz,
Beatrix Hornef,
Ute Karabek,
Sven Kolibal,
Ottmar Krämer-Fuhrmann,
Michael Kürschner,
Anastasia-Maria Leventi-Peetz,
Jörg-Volker Peetz,
Peter Post,
Cornelia Richter,
Josef Roggenbuck,
Barbara Steckel,
Angelika Weihermüller

Contact:

Klaus Wolf
Fraunhofer Institute for Algorithms and Scientific Computing SCAI
Schloss Birlinghoven
53754 Sankt Augustin
Germany
Phone +49(0)2241/14-2557
Fax +49(0)2241/14-2181
klaus.wolf@scai.fhg.de

Contents

1 About this guide **3**

2 Requirements on your code **3**

3 Requirements on the meshes at the coupling surface **3**

4 Setting up the coupling **4**

5 The MpCCI input file **4**

6 Integration of MpCCI calls into the simulation codes **9**

7 Compiling, linking, and running the coupling **24**

8 The MpCCI tracefile **27**

1 About this guide

This user's guide for the code coupling interface MpCCI gives you an introduction how to use MpCCI. It will guide you through your first steps towards multidisciplinary applications. Starting with a simple example you learn about all the steps you have to take to couple two application codes.

In this guide we will refer to the examples that are part of the MpCCI distribution (see the directory examples). Example 1 (ex1) is a very simple but complete example for coupling two sequential codes. We will use this example and example 4 to describe how to set up a coupled computation. We recommend to study these examples together with this guide very carefully because it shows the complete logic of coupling codes with MpCCI.

We will not describe all functions of MpCCI in this guide because some of them are only necessary for special applications. Nor we will explain every parameter of the discussed functions in detail. Please refer to the specification of MpCCI for such kind of information.

This user's guide gives you the basic idea of coupling codes with MpCCI and puts you on the road towards your multidisciplinary application. In case you need further assistance please contact the MpCCI-Team at mpcci@scai.fhg.de.

2 Requirements on your code

There are a few technical items which must be considered in order to integrate MpCCI into your code:

- If you have a parallel code, the internal communication between the processes of your code must be based on MPI because MpCCI is a library on the basis of MPI.
- The global communicator MPI_COMM_WORLD must not be used for the code internal communication because it also contains the processes of the remote code(s). Instead, the communicator CCI_MPI_COMM_CODE is to be used which is provided by MpCCI after calling the initialization routine CCI_Init. CCI_MPI_COMM_CODE contains all processes of the current code.

3 Requirements on the meshes at the coupling surface

Currently the standard MpCCI version is able to transfer data through a coupling surface in 3-dimensional space and through a line in the 2-dimensional space. Volume-coupling in the 3-dimensional space is supported in the advanced version of MpCCI.

The meshes on both sides of the coupling surface may not match. It is allowed that the meshes have only a partial overlap but this should be announced for performance reasons in the contact block of the MpCCI input file via the item

```
contact
  overlap = partial;
  ...
end
```

The default is that MpCCI expects overlapping coupling surfaces, i.e. for each point on the one side there should be an element on the other side where the point lies in. If your meshes only have a partial overlap region and you have not specified the mentioned parameter MpCCI will perform a linear search for the corresponding element after the bucket search algorithm failed.

A certain normal distance between the meshes at the coupling surface is allowed. Larger distances are possible, but require some settings for the neighborhood search in the input file, cf. chapter 4 of the MpCCI specification.

In the examples described here there is no distance between the meshes, only the nodes and elements differ.

4 Setting up the coupling

In the following section we briefly describe the main phases of a coupled computation and the relevant MpCCI functions. For more details, you should consult the MpCCI specification. You are also advised to study the examples in the examples subdirectory. For the beginning, please read the first example together with the following description.

Each code has to specify its coupling surface to MpCCI. In general, the coupling surface of a code is distributed among its processes. The region of a process on the coupling surface may consist of different subregions which are treated separately from each other. Such subregions are called partitions. They are the basic geometric building blocks of the coupling surface from the point of view of MpCCI, and the concept of partitions is central in MpCCI. A process specifies its partitions to MpCCI and defines coupling values in the nodes or elements of each partition. MpCCI's task is to calculate for each partition the neighborhood relations w.r.t. the mesh of the remote code and to transfer the coupling values into the nodes or elements of the remote partitions. Since in general the grids will not match at the coupling surface, some interpolation is needed for the transfer of the values. The great advantage of using MpCCI for the exchange of data instead of doing it by hand is the fact that the complicated details of the neighborhood search and interpolation are totally hidden behind the interface of the MpCCI functions. Each code sees only its own side of the coupling surface and need not care about the geometric aspects of the remote side.

To set up a coupling between codes, two tasks have to be done:

1. You must prepare a common MpCCI input file, where you define all relevant input data for the coupling, e.g. for each code the coupling quantities which it is going to exchange, the correspondence of coupling quantities for each pair of codes, the level of MpCCI output, the names and location of the binaries for each code, the number of processes for each code, the environment for the processes of each code etc.

   For simple applications the input file can also be written with the help of the MpCCI GUI.

2. You have to integrate MpCCI calls into the codes.

MpCCI supports two communication schemes:

In the following there are two examples taken from the examples directory. One example (ex1) is based on send/receive-communication, where all definitions for sending and receiving must be implemented in the code, the other example (ex4) is based on synchronization points, which requires some definitions in the MpCCI input file.

5 The MpCCI input file

The MpCCI input file consists of eight blocks where some of them are optional, some of them must not be omitted and one of them may occur several times. The order of the blocks must not be changed.

1. **code block:** For each code which is part of the coupled computation a code block must be written. If one code coupled with itself one code block is sufficient. The code block describes the quantities communicated in the coupled computation.
2. **quantities block:** Here the matching of the quantities of the different codes must be specified if several different codes are concerned. If a code is coupled with itself the quantities block may be omitted.
3. **control block:** The control block is optional. General steering parameters for `MpCCI` may be defined such as that the mesh quality must be checked or not.
4. **contact block:** The attributes of the grids, the contact search algorithm and the matching criterion may be chosen. This block is optional.
5. **switches block:** The switches block fixes how many facts are written into the outputfiles of the processes and whether a debugging output must be written. This block is optional as well.
6. **jobs block:** This block specifies the codes that participate and how they are started.
7. **parameters block:** The parameters block enables the user to provide values of different kind to the codes involved. This block is optional.
8. **coupling block:** Here synchronization points and communicators may be defined. This block is optional.
9. **add_control block:** This additional block is only mentioned for completeness. It is an `MpCCI` internal block generated in combination with the `MpCCI` GUI.

Comments are marked by a preceeding %.

The input file of the example `ex1`

We start with the input file of the example based on send/receive communication, because the input file is simpler than the input file of the other example. The comments describe the input file.

```
code simulationcode1
   % The quantities which are going to be communicated are defined
   % here: In this case there is only one quantity with the
   % following settings:
   % (1) the name     : temperature1
   % (2) an identifier: no = 7
   % (3) the location : where = node,
   %     values are taken from nodes or put into the nodes,
   %     the alternative is where = elem for element-based
   %     quantities
   % (4) the dimension: dim = scalar
   %     Another option is dim = vector
   % (5) Interpolation type: with type = field a non-conservative
   %     type is chosen.
  temperature1( no = 7, where = node, dim = scalar, type = field );
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
code simulationcode2
  temperature2( no =11, where = node, dim = scalar, type = field );
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

quantities
   % This statement is interpreted as:
   % Values for temperature1 of simulationcode1 may be communicated
   % to or from temperature2 of simulationcode2
  simulationcode1.temperature1 = simulationcode2.temperature2;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

switches
   % maximal output level
  output_level  = 3;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

jobs
   % The code underlying the job JobName1 is simulationcode1.
   % The name of the executable (exec) is ex1Binary1 and it is not
   % a parallel code for the number of processes (nprocs) is 1.
  JobName1 = simulationcode1 (
    pwd    = ".",
    exec   = "ex1Binary1",
    nprocs = 1
  );

  JobName2 = simulationcode2 (
    pwd    = ".",
    exec   = "ex1Binary2",
    nprocs = 1
  );

end
```

The input file of the example `ex4`

The example `ex4` of the `MpCCI` distribution presents three codes. Each of the codes defines two meshes as coupling surfaces. In order to simplify the communication this example uses the concept of synchronization points.

The strategy of synchronization points requires more information, but some attributes of the communication can be changed easily without new compilation of the codes.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

code simulationcode1
  quantity1( no = 1, where = node, dim = scalar, type = field );
```

```
  quantity2( no = 2, where = node, dim = scalar, type = field );
  quantity3( no = 3, where = node, dim = scalar, type = field );
  quantity4( no = 4, where = node, dim = scalar, type = field );
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

code simulationcode2
  quantityA( no = 1, where = node, dim = scalar, type = field );
  quantityB( no = 2, where = node, dim = scalar, type = field );
  quantityC( no = 3, where = node, dim = scalar, type = field );
  quantityD( no = 4, where = node, dim = scalar, type = field );
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

code simulationcode3
  quantityW( no = 1, where = node, dim = scalar, type = field );
  quantityX( no = 2, where = node, dim = scalar, type = field );
  quantityY( no = 3, where = node, dim = scalar, type = field );
  quantityZ( no = 4, where = node, dim = scalar, type = field );
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

quantities
  simulationcode1.quantity1 = simulationcode2.quantityA;
  simulationcode1.quantity2 = simulationcode2.quantityB;
  simulationcode1.quantity3 = simulationcode2.quantityC;
  simulationcode1.quantity4 = simulationcode2.quantityD;
  simulationcode1.quantity1 = simulationcode3.quantityW;
  simulationcode1.quantity2 = simulationcode3.quantityX;
  simulationcode1.quantity3 = simulationcode3.quantityY;
  simulationcode1.quantity4 = simulationcode3.quantityZ;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

switches
  output_level  = 3;    % maximal output level
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

jobs

  JobName1 = simulationcode1 (
    pwd    = ".",
    exec   = "ex4Binary1",  % name of the executable
    nprocs = 1              % number of processes
  );

  JobName2 = simulationcode2 (
    pwd    = ".",
```

```
    exec   = "ex4Binary2",
    nprocs = 1
  );

  JobName3 = simulationcode3 (
    pwd    = ".",
    exec   = "ex4Binary3",
    nprocs = 1
  );
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

coupling

   %
   %  For job "JobName1" one synchronization point with local id 1 is
   %  specified: At CCI_Reach_sync_point( 1, ... ) job "JobName1" sends
   %  values of quantity1 on mesh 2 and values for quantity3
   %  on mesh 1 and receives values for quantity4 on mesh 2.
   %
   %  For JobName2 two synchronization points are specified:
   %  At both synchronization points ( with identifier 2 and 1 )
   %  JobName2 sends values of quantityD on mesh 2 and
   %  receives quantityA on mesh 2 and quantityB on mesh 1.
   %
   %  For JobName3 two different synchronization points are specified:
   %  At synchronization point 1 quantityX on mesh2 will be sent
   %  and quantityY on mesh2 will be received. At synchronization
   %  point 3 quantityX on mesh 2 and quantityW on mesh 1 will be sent
   %  and quantityZ on mesh 2 will be received.
   %
  syncpt JobName1( 1 ) : send( quantity1/2, quantity3/1 ),
                         recv( quantity4/2);
  syncpt JobName2( 2 ) : send( quantityD/2 ),
                         recv( quantityA/2, quantityB/1 );
  syncpt JobName3( 1 ) : send( quantityX/2 ),
                         recv( quantityY/2 );

  syncpt JobName3( 3 ) : send( quantityX/2, quantityW/1 ),
                         recv( quantityZ/2 );
  syncpt JobName2( 1 ) : send( quantityD/2 ),
                         recv( quantityA/2, quantityB/1 );

   %
   %  The following statement is interpreted in the following way:
   %  Synchronization point 1 of JobName1 is matched with
   %  synchronization point 2 of JobName2 and synchronization point 1
   %  of JobName3. Synchronization point 1 of JobName2 matches with
   %  synchronization point 3 of JobName3. There is no matching
   %  synchronization point specified for JobName1.
   %
  match_syncpt   JobName1( 1, - )
              = JobName2( 2, 1 )
```

```
                = JobName3( 1, 3 );
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

6 Integration of MpCCI calls into the simulation codes

The coupled computation has four phases:

1. Initialization phase
2. Mesh definition phase
3. Communication phase
4. Finalization phase

The phases must be implemented in the above order.

In the initialization phase the code is prepared for the coupled communication by the MpCCI-subroutine CCI_Init. MPI is initialized and the predefined communicators are defined here . If e.g. MPI is used in the simulation code for parallel computation and both calls MPI_Init and CCI_Init are necessary then MPI_Init must be called before CCI_Init. Instead of CCI_Init the subroutine CCI_Init_with_id_string may be used. In addition to the initialization tasks this routine overwrites the id_string which can be given in the jobs block of the input file.

In the mesh definition phase the geometry of the grid must be made known to MpCCI:

1. The identifier and names of a mesh, optionally (CCI_Def_mesh)
2. The partitions of the mesh which are treated e.g. by different processes (CCI_Def_partition)
3. The nodes of the partitions (CCI_Def_nodes)
4. The elements of the partitions based on the nodes (CCI_Def_elems). This may be omitted if nearest_neighbor interpolation is specified in the input file.
5. The matching of a partition with partitions of the remote code. (This may be omitted, but useful in many cases.)

These definitions must be integrated in the above order.

The second task in this phase is to supply the code with information from the input file.

- Specification of synchronization points
  (CCI_Sync_points_info and CCI_Def_sync_points)
- Specification of communicators
  (CCI_Comm_info and CCI_Def_comm)
- Specification of parameters (CCI_Parameter_info)
- Specification of quantity properties and mesh properties (CCI_General_info)
- Specification of code and job name (CCI_Get_id_string)

The mesh definition phase is completed when CCI_Close_setup is called. This MpCCI-subroutine must be called because the neighborhood search and some preparations relevant for the data communication are done here. The mesh quality is checked if it is not turned off by a switch in the input file.

After the call of CCI_Close_setup the communication phase starts. Values on nodes or elements and global values can be transfered now. One communication step consists of the following parts:

1. The values must be prepared for sending for each quantity on each partition (CCI_Put_nodes and CCI_Put_elems)
2. The values for several quantities on several meshes must be send by one code and received by another. There are two communication schemes implemented in MpCCI
   - Based on CCI_Send (or CCI_Isend) and CCI_Recv where quantities, meshes and communicators are to be specified in the code.
   - Based on synchronization points CCI_Reach_sync_point which can be specified in the input file. The big advantage of the scheme lies in the fact that changes of the quantities to be sent or received do not require a compilation of the code.
3. The values must be obtained by the receiver code for each quantity on each partition.(CCI_Get_nodes and CCI_Get_elems)

To exchange information between the codes about the state of the simulation the subroutine CCI_Check_convergence may be used during the communication phase.

The finalization phase consists of at most two statements: The subroutine CCI_Finalize terminates the coupling. If MPI_Init was called MPI_Finalize must be called after CCI_Finalize.

The codes for example 1

The example based on CCI_Send and CCI_Recv is now described. The description of the initialization phase, the mesh definition phase and the finalization phase are the same for both examples.

Initialization phase

```
-> MPI_Init()
-> CCI_Init()
```

The MPI and MpCCI instances of the process are initialized. For each code, a communicator CCI_MPI_COMM_CODE is built which comprises all processes of the code. Furthermore, the environment of the process (working directory, environment variables) is defined as specified in the MpCCI input file (see below).

If your code has not already called MPI_Init when calling CCI_Init or if your code is a sequential code and does not call MPI_Init at all you don't have to do it. CCI_Init checks if MPI is initialized and will call MPI_Init itself if necessary.

Mesh definition phase

Each MpCCI process defines the geometric data for all its partitions. First, the partition itself has to be defined for the corresponding mesh:

```
-> CCI_Def_partition ( meshId, partId )
```

Then the node coordinates of each partition are committed to MpCCI:

```
-> CCI_Def_nodes( meshId, partId, ..., nNodes,
                                  ..., coordinates )
```

Note, that the process treats each partition separately from the others, in the mesh definition phase as well as later in the coupled computation phase.

After defining the node coordinates, the process defines the elements of each partition and thereby its topology:

```
-> CCI_Def_elems( meshId, partId, nElems, ..., elemNodes )
```

Also the element types are specified. If there are several element types on the partition, you can specify for each element the type in the array elemTypes. Currently the following element types are supported: triangles with 3 and 6 datapoints, quadrilaterals with 4, 8 or 9 data points, pentagons and hexagons for surface coupling, tetrahedrons, hexahedrons, pyramids and prisms for volume coupling and line segments for line coupling.

After the process has defined all geometric data for each of its partitions, it closes the mesh definition phase by calling

```
-> CCI_Close_setup( switch )
```

This call initiates MpCCI to perform the neighborhood calculation which is needed for the subsequent exchange of coupling data.

With this last MpCCI initialization routine a process declares whether or not it is involved in the coupling surface: in the former case it commits switch = 1, in the latter case switch = 0. Processes which are involved in the coupling surface are called MpCCI processes. Only MpCCI processes are relevant for the coupling, the other processes do not take part in the MpCCI communication. They must not call any further MpCCI routines with the exception of CCI_Finalize.

Communication phase

Now, after committing all relevant geometric data to MpCCI, the central part of the coupling can start: The exchange of coupling quantities between the codes through the coupling surface.

On the sender side, each MpCCI process must define in all its partitions the values of all relevant coupling quantities which are to be sent to the other code:

```
-> CCI_Put_nodes( meshId, partId, quantityId, ..., values )
```

(or CCI_Put_elems if the values refer to the elements of the partition rather than its nodes).

quantityId is the code local id for a quantity which is to be exchanged.

After defining in each partition the values of the relevant coupling quantities, the process sends all relevant quantities to the remote side. This can be done either via a blocking call:

```
-> CCI_Send( nQuantityIds, quantityIds, ..., comm )
```

(i.e. the process waits until the corresponding CCI_Recv on the remote side has been called) or via a non-blocking call:

```
-> CCI_Isend( nQuantityIds, quantityIds, ..., comm, request )
```

The non-blocking communication must be completed by calling `CCI_Wait( request, status )` later otherwise memory leaks would occur.

When calling `CCI_Send` or `CCI_Isend` also a `MpCCI` communicator has to be specified. If only one mesh is used within the coupled communication the predefined `MpCCI` communicator `CCI_COMM_RCODE( remoteCodeId )` can be used. Otherwise a `MpCCI` communicator has to be defined by calling `CCI_Def_comm`. For detailed information please refer to the `MpCCI` specification or example 3 in the subdirectory `ex3`.

On the receiver side, each `MpCCI` process must call:

```
-> CCI_Recv( nQuantityIds, quantityIds, ..., comm, status )
```

Here, `nQuantityIds` and `quantityIds` define the quantities to be received by this call. If `nQuantityIds` is set to zero, quantities will be received in the same order as they have been sent. The identifiers of the received quantities are stored in the status array.

The values of the received quantities are stored internally by `MpCCI` and can be read by the call:

```
-> CCI_Get_nodes( meshId, partId, quantityId, ..., values )
```

or `CCI_Get_elems` if the values refer to the elements rather than to the nodes.

These four calls `CCI_Put_nodes` and `CCI_Send` on the sender side, `CCI_Recv` and `CCI_Get_nodes` on the receiver side are the basic routines which are used for the exchange of the coupling quantities. They reflect the intuitive picture of this exchange: "Put values into the transport medium on the one side and initiate the transport there. On the other side, initiate the receiving of the values and get them out of the transport medium."

If synchronization points are used `CCI_Send` and `CCI_Recv` must be replaced by `CCI_Reach_sync_point`.

Finalization phase

```
-> CCI_Finalize()
-> MPI_Finalize()
```

Clean up the `MpCCI` and MPI instances of each process. Even non-`MpCCI` processes must call these routines. If `MPI_Init` was called by `CCI_Init` `MPI_Finalize` will be called by `CCI_Finalize`.

Code 1 of example 1 (C version):

```
#include "cci.h"
#include <stdio.h>

/*%%%
```

```
  *         |        |        |
  *         |4       |5       |6
  *         *--------*--------*
  *         |        |        |
  *         |        |        |
  *         |1       |2       |3
  *         *--------*--------*
  */
static float NODEDATA[] =  { 0,    0,    0,
                             0.5,  0,    0,
                             1,    0,    0,
                             0,    0.5,  0,
                             0.5,  0.5,  0,
                             1,    0.5,  0,
                             0,    1,    0,
                             0.5,  1,    0,
                             1,    1,    0 };

static int TOPOLOGYDATA[]  = { 1, 2, 5, 4,
                               2, 3, 6, 5,
                               4, 5, 8, 7,
                               5, 6, 9, 8 };

/*%%%
```

```
  /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*
   *  (I) Initialization phase  *
   *%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

  /*  (I.1) Open MPI and MpCCI instance:  */

  CCI_Init( &argc, &argv );

  printf( "I am a C code.\n" );

  /*%%%%%%%%%%%%%%%%%%%
```

```
  for ( j = 0; j < NNODES; j++ ) valueData[j] = j;

  /*  (III.1) Put the values into MpCCI (local operation):  */

  quantityId   = 7;
  quantityDim  = 1;

  CCI_Put_nodes( meshId, partId, quantityId, quantityDim,
                 NNODES, nNodeIds, nodeIds, REALTYPE, valueData );

  /*  (III.2) Send these values to the remote code:  */

  nQuantityIds    = 1;
  quantityIds[0]  = 7;
  nLocalMeshIds   = 1;
  localMeshIds[0] = 1;
  remoteCodeId    = 2;
  comm            = CCI_COMM_RCODE[remoteCodeId];

  CCI_Send( nQuantityIds, quantityIds,
            nLocalMeshIds, localMeshIds, comm );

/*%%%
```

```
  *         *------*----------*
  *         |   /  |      /   |
  *        1| /   2|  /      3|
  *         *------*----------*
  */
static float NODEDATA[] = { 0,     0,     0,
                            0.4,   0,     0,
                            1,     0,     0,
                            0,     0.4,   0,
                            0.4,   0.4,   0,
                            1,     0.4,   0,
                            0,     1,     0,
                            0.4,   1,     0,
                            1,     1,     0 };

static int TOPOLOGYDATA[] = { 1, 5, 4,
                              1, 2, 5,
                              2, 6, 5,
                              2, 3, 6,
                              4, 8, 7,
                              4, 5, 8,
                              5, 9, 8,
                              5, 6, 9 };

/*%%%
```

```
int nEmptyNodes;

float valueData[NNODES];

/*%%%
```

```
/*%%%
```

```
int main( int argc, char** argv )
{
  /*  Variables for both meshes: */
  int globalDim = 3;
  int realType  = CCI_FLOAT;

/*%%%
```

```
/*%%%
```

```
  int sendMeshes1[CCI_MAX_NQUANTITIES];
  int sendnQuantities1;
  int recvQuantities1[CCI_MAX_NQUANTITIES];
  int recvMeshes1[CCI_MAX_NQUANTITIES];
  int recvnQuantities1;

  int j;

/*%%%
```

```
   *%%%
```

```
   *  Before reaching the synchronization point, we put values
   *  in the
   *  CCI_Put_nodes for synchronization point 1:
   */
  {
    int         quantityDim     = 1;
    float       values1[]       = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
                /* Same values for all quantities */

    for ( j = 0; j < sendnQuantities1; ++j )
    {
      if ( sendMeshes1[j] == meshIdMesh1 )
      {
        /*  Mesh 1:  */
        CCI_Put_nodes( meshIdMesh1, partIdMesh1, sendQuantities1[j],
                       quantityDim, nNodesMesh1, nNodeIdsMesh1,
                       nodeIdsMesh1, realType, values1 );
      }
      else
      {
        /*  Mesh 2:  */
        CCI_Put_nodes( meshIdMesh2, partIdMesh2, sendQuantities1[j],
                       quantityDim, nNodesMesh2, nNodeIdsMesh2,
                       nodeIdsMesh2, realType, values1 );
      }
    }
  }


/*%%%
```

```
    for ( j = 0; j < recvnQuantities1; ++j )
    {
      if ( recvMeshes1[j] == meshIdMesh1 )
      {
        /*  Mesh 1:  */
        CCI_Get_nodes( meshIdMesh1, partIdMesh1, recvQuantities1[j],
                       quantityDim, nNodesMesh1, nNodeIdsMesh1,
                       nodeIdsMesh1, realType, valuesMesh1,
                       maxnEmptyNodes, emptyNodes, &nEmptyNodes );
      }
      else
      {
        /*  Mesh 2:  */
        CCI_Get_nodes( meshIdMesh2, partIdMesh2, recvQuantities1[j],
                       quantityDim, nNodesMesh2, nNodeIdsMesh2,
                       nodeIdsMesh2, realType, valuesMesh2,
                       maxnEmptyNodes, emptyNodes, &nEmptyNodes );

      }
    }
  }


/*%%%
```

Examples:

Your code is a Fortran code and contains the modules `mod1.f` and `mod2.f`. Then, the binary with name `myCode` is produced in the following way:

```
ccif77 -c mod1.f mod2.f
ccilink mod1.o mod2.o -o myCode
```

If `mod1.f` and `mod2.f` are Fortran 90 modules you have to use

```
ccif90 -c mod1.f mod2.f
ccilink -f90 mod1.o mod2.o -o myCode
```

to compile and link your program with `MpCCI`.

If your code is a C code like for instance the second example you may proceed as follows if the three codes are `code1C.c`, `code2C.c`, and `code3C.c` and the input file is `ex4cci`:

```
ccicc -c code1C.c
ccicc -c code2C.c
ccicc -c code3C.c

ccilink code1C.o -o ex4Binary1C
ccilink code2C.o -o ex4Binary2C
ccilink code3C.o -o ex4Binary3C

ccirun -nocontrol ex4.cci
```

To start your coupled computation you have to use the ccirun command. ccirun will read the `MpCCI` input file and start the listed executables with mpirun. ccirun supports several options:

```
ccirun  - script for starting the coupled computation.
          Usage: ccirun [opts] <mpcci_input_file>

          The following options are supported:

          -debug <debug_file>  Run some processes under a
                               debugger, cf.
                               $CCI_HOME/doc/README.debug

          -display <DISPLAY>   In case of -xterm or -xask,
                               start the X terminals on the
                               given display <DISPLAY>.

          -hostlist <hostlist_file>
                               Specify a different hostlist
                               file (from that defined by
                               the environment variable
                               CCI_HOSTLIST)

          -includepath <path>  Set search path for MpCCI include
                               files (default is defined by the
```

```
                    environment variable CCI_PATH)

-log                Write a separate output file
                    for each process. The file
                    produced by the process with
                    global rank <r> will be
                    ccirun.out.<r> (ccirun.out.0
                    will be the output of the
                    control process.)

-mpiopt <>          Specify additional options
                    which are passed to mpirun

-nocontrol          Run without a control process

-nolocal            Process 0 does not run on the
                    local host, but on the first host
                    of the hostlist

-nonfs              Specify this option if there
                    is no common file system. No
                    checks of existence of
                    files/directories specified
                    in the MpCCI input file are
                    done

-norun              All preparation tasks are done,
                    but the coupled computation
                    is not started.

-noscripts          The executables specified in the
                    MpCCI input file are started
                    directly instead of writing
                    scripts which in turn start the
                    executables. The latter is the
                    default and has some advantages
                    for separating the output of the
                    different processes.

-o <outfile>        Specify a name for the stem of
                    the output file. The output
                    file for a process with global
                    rank <r> will be <outfile>.<r>
                    (default: ccirun.out)

-observe            Observe all processes in separate
                    X terminals
                    (via 'tail -f <outputfile>')

-path <path>        Set search path for executables
                    (default is defined by the
                    environment variable PATH)

-preview <tracefile> For debugging only. Skips the
```

```
                        MpCCI internal neighborhood
                        search in CCI_Close_setup,
                        writes all geometry data into
                        the given tracefile, and exits
                        the coupled computation. The
                        tracefile can be used to
                        visualize the meshes in
                        the coupling region.

-pwd <PWD>              Working directory of ccirun

-tmpdir <dir>           Set a directory for temporary
                        files.
                        The default is defined by the
                        environment variable CCI_TMPDIR
                        or (if this is not set) '/tmp'.

-version                Print out the version of
                        MpCCI used

-xask                   Ask the user for each process
                        using a popup menu how it
                        should be run
                        (silent, xterm, debug)

-xterm                  Open a separate X terminal for
                        each process and write the stdout
                        of the process into that Xterm.
                        In this way you "see" the
                        computation running.
```

If any other option is specified (i.e. -h/-help) `ccirun` will print out some help information.

A typical sequence of calling the `MpCCI` scripts for compiling, linking, and running is shown for the examples of the distribution. You can study the makefiles or just type

```
make -n runF
make -n runC
make -n runCxx
```

in the example directories `ex1`, `ex2`, `ex3`, or `ex4`. You will get a list of all calls performed by the make command for the three different programming languages Fortran, C, and C++.

8 The `MpCCI` tracefile

The `MpCCI` tracefile is an output file produced by the `MpCCI` control process during the coupled computation. This file enables the user to take a closer look

- at the geometries of the codes defined to `MpCCI` and
- at the coupling algorithm, i.e. the sends and receives

with a suitable visualization tool like **ccivis**. Thus, the `MpCCI` tracefile is a part of a debugging tool and helps the user to understand what is going on within the coupling.

What to do to get an MpCCI tracefile?

There are some pre-conditions that have to be fulfilled in order to get an MpCCI tracefile. Although this requirements are simple they are named here to ensure the tracefile output:

- First, the name of the tracefile has to be given. This is done in the control block of the MpCCI input file (see the MpCCI specification).
- Second, MpCCI has to be started with the MpCCI control process. This process collects the output of all processes and writes it to a file with the given name. So, please do not call ccirun with the option no_control! Otherwise, you will start MpCCI without a control process and the rule is: no control process = no tracefile.
- Another point to mind is the usage of restart files. If restart files are used in mode read some neighborhood informations might be missing for the tracefile, and so, the tracefile will not be written in this case. (This pre-condition might be dropped in the future.)
- When setting up the coupling, the user sometimes just wants to see if the meshes are defined correctly. MpCCI supports this with the ccirun option -preview <tracefilename>. In this case, all geometry data are written into the given tracefile, the time-consuming neighborhood search is left out and the coupled computation is stopped. The tracefile can be used then to visualize the meshes in the coupling region.

Contents of the MpCCI tracefile

The MpCCI tracefile keeps trace of the coupled computation. So, the file contains

- the geometries of the codes defined to MpCCI, i.e the definitions of the meshes, the partitions, the nodes and elements. With a suitable visualization tool like **ccivis** the user will be able to look at all of these parts and check the correctness, e.g. if there are orphaned points.
- the sends and receives of the coupling algorithm and the values communicated. So the user can see which values have been sent and which values have been received. As an improvement, the current version supports the exchange of global values.

Improvements of the MpCCI tracefile

The visualization tool **ccivis** (as the former tool **COCOVis**) is prepared to show the sends and receives in an animation based on **couplings steps**. These steps have to be initiated by the user otherwise all sends and receives are collected in one step which forms a rather poor animation. So, what is necessary to create couplings steps?

The user has to identify the coupling step(s) within his or her coupling algorithm. This can be, e.g. a time step, or a block between two synchronization points. Now, a call of **CCI_Check_convergence** has to be inserted in the codes, and thats all.

```
/*%%%
```

```
  while ( ! couplingNotAtEnd )
  {
     /**
      *   all that has to be done during one coupling step
      */
     ...
     CCI_Send(....);   // could also be: CCI_Recv

     /**
      *   create next coupling step
      */
     CCI_Check_convergence( CCI_CONTINUE, &globalConvergence, CCI_ANY_CODE);

     couplingNotAtEnd = ...;
  }

/*%%%
```